We are IntechOpen,
the world's leading publisher of
Open Access books
Built by scientists, for scientists

6,900
Open access books available

186,000
International authors and editors

200M
Downloads

154
Countries delivered to

Our authors are among the
TOP 1%
most cited scientists

12.2%
Contributors from top 500 universities

CLARIVATE ANALYTICS
BOOK CITATION INDEX
INDEXED

WEB OF SCIENCE™
Selection of our books indexed in the Book Citation Index
in Web of Science™ Core Collection (BKCI)

Interested in publishing with us?
Contact book.department@intechopen.com

Numbers displayed above are based on latest data collected.
For more information visit www.intechopen.com

Concurrent Specification of Embedded Systems: An Insight into the Flexibility vs Correctness Trade-Off

F. Herrera and I. Ugarte
University of Cantabria
Spain

1. Introduction

In 2002, (Kish, 2002) warned about the danger of the abrupt break in Moore's law. Fortunately, nowadays integration capabilities are still growing and 20nm and 14nm technologies are envisaged, (Chiang, 2011). However, the frequency of integrated circuits cannot grow anymore. Therefore, in order to achieve a continuous improvement of performance, computer architectures are evolving towards the integration of more and more parallel computing resources. Examples of this include modern Graphical Processing Units (GPUs), such as the new CUDA architecture, named Fermi, which will use 512 cores, (Halfhill, 2012). Embedded system architectures show a similar trend with General Purpose Processors (GPPs), and some mobile phones already included between 2 and 8 RISC processors a few years ago, (Martin, 2006). Moreover, many embedded architectures are heterogeneous, and enclose different types of truly parallel computing resources such as (GPPs), Co-Processors, Digital Signal Processors, GPUs, custom-hardware accelerators, etc.

The evolution of HW architectures is driving the change in the programming paradigm. Several languages, such as (OpenMP, 2008), and (MPI, 2009), are defining the de facto programming paradigm for multi-core platforms. Embedded MPSoC platforms, with a growing number of general purpose RISC processors, are necessitating the adoption of a task-level centric approach in order to enable applications which efficiently use the computational resources provided by the underlying hardware platform.

Parallelism can be exploited at different levels of granularity. GPU-related languages enable the handling of a finer level of granularity, in order to exploit the inherent data parallelism of graphical applications. These languages also enable some explicit handling of the underlying architecture. MPSoC homogenous architectures require and enable a task-level approach, which provides a larger granularity in the handling of concurrency, and a higher level of abstraction to hide architectural details. A task-level approach enables the acceleration problem to be seen as a partition of functionality into *tasks* or high-level processes. A standard language which enables a task-level specification of concurrent functionality, and its communication and synchronization is convenient. In this scenario, SystemC (IEEE, 2005) standard has become the most widespread language for the specification of embedded systems. The main reason is that SystemC extends C/C++ with a set of features for a rich, standard modelling of concurrency, time, data types and modular hierarchical.

Summing up, concurrency is becoming a must in embedded system specification as it has become necessary for exploiting the underlying concurrency of MPSoC platforms. However, it brings a higher degree of complexity which introduces new challenges in embedded system specification, (Lee, 2006). In this chapter, the challenges and solutions for producing concurrent and correct specifications through simulation-based verification techniques are reviewed, and an alternative based on correct-by-construction specification methodologies is introduced. The chapter mainly addresses abstract concurrent specifications formed by asynchronous processes (formally speaking, untimed models of computation, MoCs, (Jansch, 2004). This type of modelling is required for speeding up the simulation of complex systems in new design activities, such as Design Space Exploration (DSE). This chapter does not assume a single definition of "correct" specification. For instance, functional determinism can be required or not, depending on the application and on the intention of the specification. However, to check whether such a property is fulfilled for every case requires the provision of the means for considering the different execution paths enabled by the control statements of an initially sequential algorithm, and, moreover, for considering the additional paths raised by a concurrent partition of such an algorithm.

The chapter will review different approaches and techniques for ensuring the correctness of concurrent specifications, to finally establish the trade-off between the flexibility in the usage of a specification language and the correctness of the coded specification. The rest of the chapter is structured as follows. Section 2 introduces an apparently simple specification problem in order to show how a rich specification language such as SystemC enables many different correct solutions, but also similar incorrect ones. Then, section 3 explores the possibilities and limitations of checking a SystemC specification through the application of simulation-based verification techniques. Finally, section 4 introduces an alternative, based on methodologies for correct-by-construction specifications and/or specification for verification. Section 5 gives conclusions about the trade-off between specification flexibility and verification cost and feasibility.

2. A "simple" specification problem

Some users may identify the knowledge of a specification language with the specification methodology itself. These users will take for granted that knowing the syntax, semantics and grammatical rules of the language is enough to build a "correct", or suitable, specification for a given design flow. Later on, in section 3, the benefits of this will be discussed. For now, let's see how a specification problem can be tackled in different ways.

A rich language provides great flexibility to tackle a similar specification problem in different ways, which in many cases is seen as a benefit by designers. In this sense, a simple experiment enabled the authors to deduce that this richness is actually employed when different users tackle the same specification problem. Let's assume we want to build a specification able to solve the functionality sketched in Fig.1.

This functionality is summarized by the following equations:

$$y = f_Y(a,b) = f_{12}\left(f_{11}(a), f_{21}(b) \right) \quad (1)$$

$$z= f_Z(a,b)= f_{22}(f_{11}(a), f_{21}(b)) \tag{2}$$

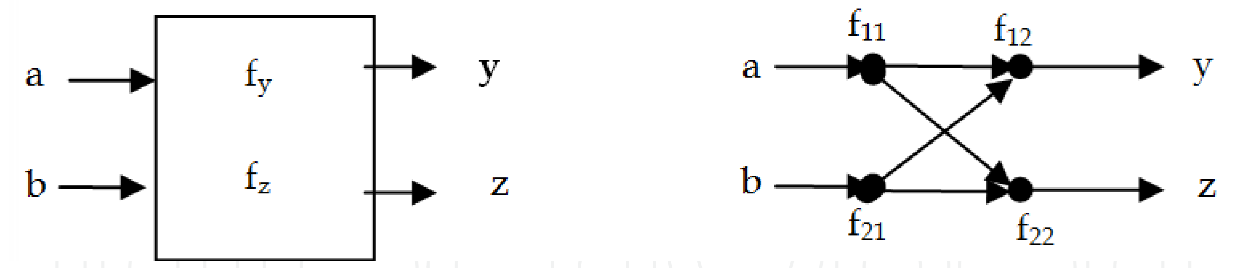


Fig. 1. Specification Intent.

In principle, the specification problem posed in Fig.1 is sufficiently general and simple to enable reasoning about it. The simple set of instances of f_{ij} functionalities, given by equation (3) will be used later on for facilitating the explanation of examples. However, the same reasoning and conclusions can be extrapolated to heavier and more complex functionalities.

$$f_{11}(x)= x+1 \quad f_{21}(x)= x+2 \tag{3}$$

$$f_{12}(x_1, x_2)= x_1+ x_2 \quad f_{22}(x_1, x_2) = (x_1=25,713)?\ 2x_1-x_2+5 :\ x_2- x_1$$

Initially, this is a straightforward specification problem which can be solved with a sequential specification, e.g., written in C/C++. The only condition to be fulfilled is to obey the dependency graph among f_{ij} functionalities shown on the right hand side of Fig.1. Thus, for instance, if the program executes the sequence $\{f_{11}, f_{21}, f_{12}, f_{22}\}$, it will be considered a correct model, and the model will produce its corresponding output as expected. For example, for (a,b)=(1,2), an output (y,z) = (6,2), where $f_{11}(1)=2$, $f_{21}(2)=4$, $f_{12}=2+4=6$ and $f_{22}=4-2=2$ (since $x_1=2\neq25,713$). Here, a user will already find some flexibility, once the order of f_{ij} executions can be permuted without impact on the intended functionality. Things start to get more complex when concurrency enters the stage. Once a pair of functionalities f_{ij} and f_{mn} can run concurrently no assumption about their execution order can be made. Assuming an atomic execution (non-preemptive) of f_{ij} functions, the basic principle for getting a solution fulfilling the specification intent of Fig. 1 is to guarantee the fulfilment of the following conditions:

$$T(f_{12}) > T(f_{11}) \tag{4}$$

$$T(f_{12}) > T(f_{21}) \tag{5}$$

$$T(f_{22}) > T(f_{21}) \tag{6}$$

$$T(f_{22}) > T(f_{11}) \tag{7}$$

Where $T(f_{ij})$ stands for the time tag associated with the computation of functionality f_{ij}. Equations (4-7) are conditions which define a partial order (PO) in the execution of f_{ij} functionalities. It is a partial order because it defines an execution order relationship only for a subset of the whole set of pairs of f_{ij} functionalities. In other words, there are pairs of functionalities, f_{ij} and f_{mn}, with $i\neq m \vee j\neq n$, which do not have any order relationship. This no order relationship is denoted $f_{ij} >< f_{mn}$. Some specification methodologies, such as HetSC, help the designer capture untimed specifications, which implicitly capture a PO. Untimed specifications reflect conditions only in terms of execution order, without assuming specific physical time conditions, thus they are the most abstract ones in terms of time handling. The PO is sufficient for ensuring the same specific global system functionality, while it reflects the available flexibility for further design steps. Indeed, no-order relationships spot functionalities which can be run in natural parallelism (that is, they are functionalities which do not require pipelining for running in actual parallelism) or which can be freely scheduled.

SystemC has a discrete event (DE) semantics, which means that the time tag is twofold, that is, T=(t, Δ). Any computation or event happens in a specific delta cycle (Δ_i). Additionally, each delta has an associated physical time stamp (t_i), in such a way that a set of consecutive deltas can share the same time stamp (this way, instantaneous reactions can be modelled as reactions in terms of delta advance, but no physical time advance). Complementarily, it is possible that two consecutive delta cycles present a jump in physical time ranging from the minimum to the maximum physical time which can be represented.

Since SystemC provides different types of processes, communication and synchronization mechanisms for ensuring the PO expressed by equations (4-7), it is easy to imagine that there are different ways to solve the specification intent in Fig.1 as a SystemC concurrent specification, even if only untimed specifications are considered. In order to check how such a specification would be solved by users knowing SystemC, but without knowledge of particular specification methodologies or experience in specification, six master students were asked to provide a concurrent solution. No conditions on the use of SystemC were set.

Five students managed to provide a correct solution. By "correct" solution it is understood that for any value of 'a' and 'b', and for any valid execution (that is, fulfilling SystemC execution semantics) the output results were the expected ones, that is $y=f_Y(a,b)$ and $z=f_Z(a,b)$. In other words, we were looking for solutions with functional determinism, (Jantsch, 2004). A first interesting observation was that, from the five correct solutions, four different solutions were provided. These solutions were considered different in terms of the concurrency structure (number of processes used, which functionality is associated to each process), communication and synchronization structure (how many channels, events and shared variables are used, and how they are used for process communication), and the order of computation, communication and synchronization within a process.

Fig. 2, 3 and 4 sketch some possible solutions where functionality is divided into 2 or 4 processes. These solutions are based on the most primitive synchronization facilities provided by SystemC ('wait' statements and SystemC events), using shared variables for data transfer among functionalities. Therefore, the solutions in Fig. 2, 3 and 4 reflect only a subset of the many coding possibilities. For instance, SystemC provides additional specification facilities, e.g. standard channels, which can be used for providing alternative solutions.

Fig.2, Fig.3a and Fig.3b show two-process-based solutions. In Fig. 2, the two processes P1 and P2 execute f_{i1} functionalities before issuing a wait(d) statement, with d of 'sc_time' type and where 'd' can be either a single delta cycle delay (d=SC_ZERO_TIME) or a timed delay (s>SC_ZERO_TIME), that is, an advance of one or more deltas (Δ) with an associated physical time advance (t). Notice that this actually means two different solutions in SystemC, under the SystemC semantics. In the former case, f_{11} and f_{21} are executed in Δ_0,

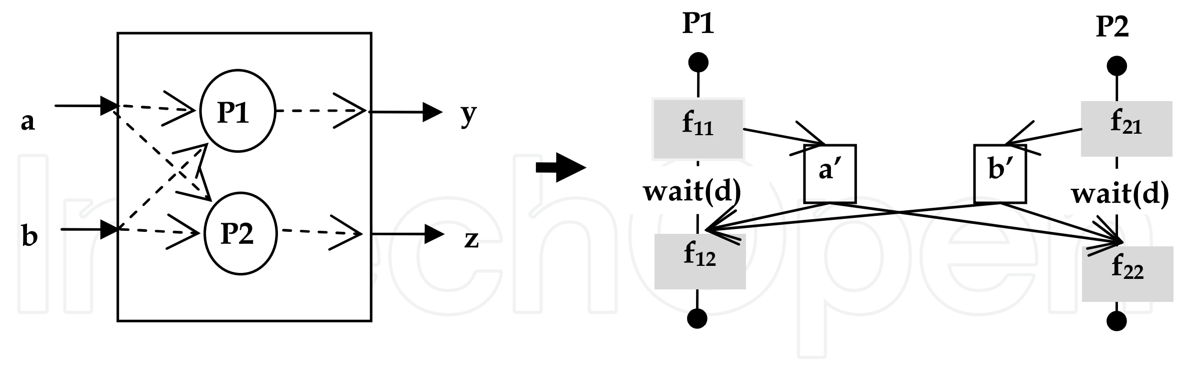


Fig. 2. Solution based on two processes and on wait statements.

while f_{21} and f_{22} are executed in Δ_1, without t advance, while in the latter case, f_{21} and f_{22} are executed in a T with a different t coordinate. Anyhow, in both cases the same untimed and abstract semantics is fulfilled, in the sense that both fulfil the same PO, that is, equations (4-7) are fulfilled. Notice that there are more solutions derived from the sketch in Fig. 2. For instance, several 'wait(d)' statements can be used on each side.

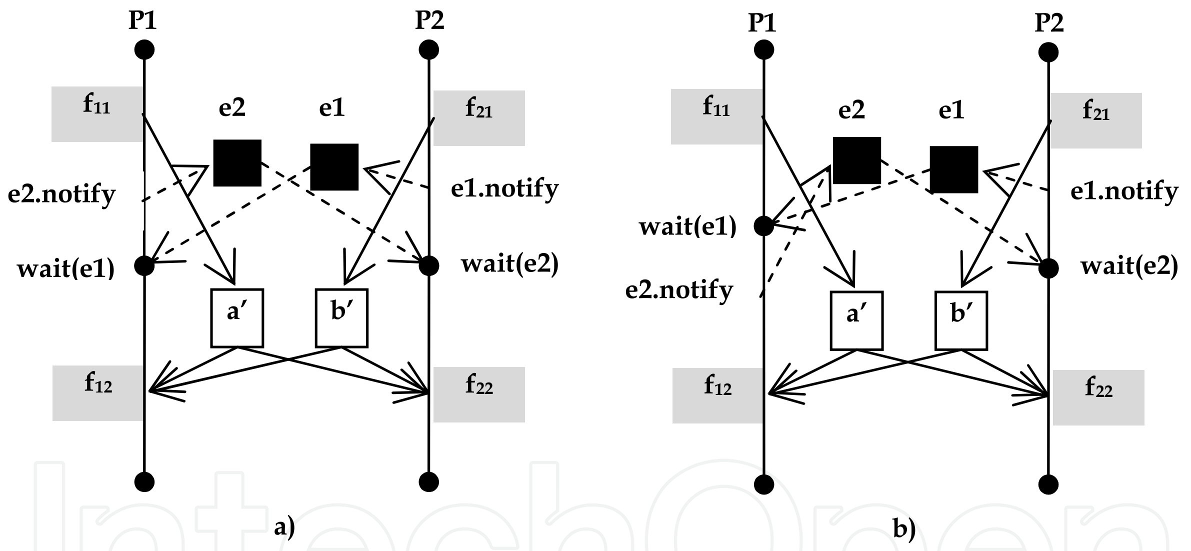


Fig. 3. Solutions based on two processes and on SystemC events.

Fig.3a and Fig.3b show two solutions based on SystemC events. In the Fig.3a solution, both processes compute f_{11} and f_{21} in Δ_0 and schedule a notification to a SystemC event which will resume the other process in the next delta. Then, both processes get blocked. The crossed notification sketch ensures the fulfilment of equations (5) and (7). Equations (4) and (6) are fulfilled since f_{11} and f_{12} are sequentially executed within the same process (P_1), and similarly, f_{21} and f_{22} are sequentially executed by process P_2. Notice that several variants based on the Fig.3a sketch can be coded without impact on the fulfilment of equations (4-7). For instance, it is possible to use notifications after a given amount of delta cycles, or after physical time and still fulfil (4-7). It is also possible to swap the execution of f_{11} and e_2 notification, and/or to swap the execution of f_{11} and e_1 notification.

Fig.3b represents another variant of the Fig.3a solution where one of the processes (specifically P_1 in Fig.3b) makes the notification after the wait statement. It adds an order condition, described by the equation $T(f_{22}) > T(f_{12})$, and which obliges the execution to require one delta cycle more (f_{22} will be executed in a delta cycle after f_{12}). Anyhow, this additional constraint on the execution order still preserves the partial order described by equations (4-7) and guarantees the functional determinism of the specification represented by Fig. 3b.

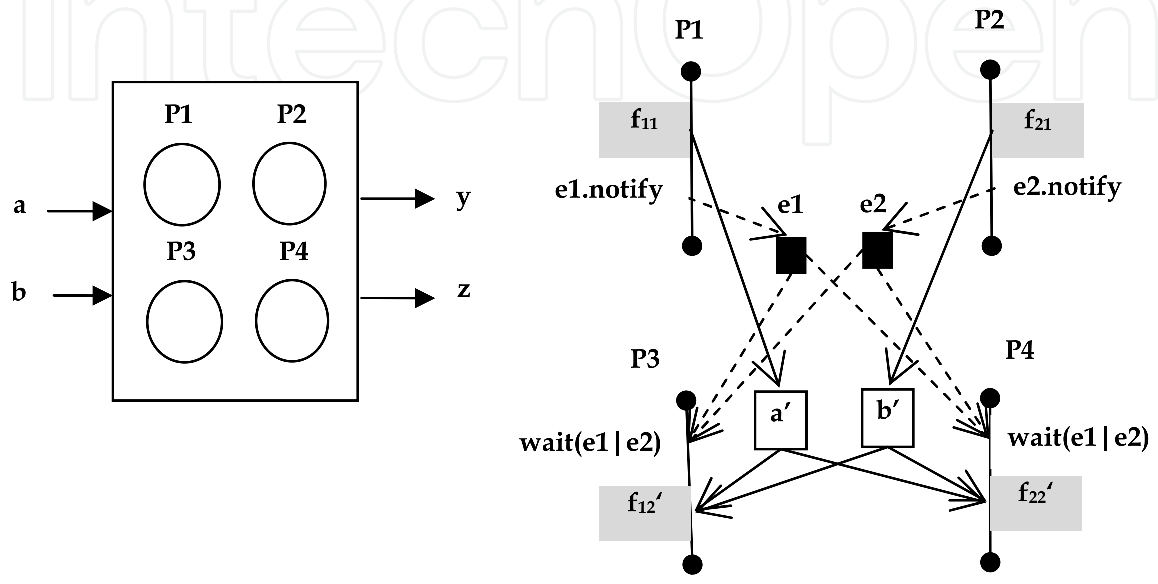


Fig. 4. Solution based on four finite and non-blocking processes.

Finally, Fig.4 shows a solution with a higher degree of concurrency, since it is based on four finite non-blocking processes. In this solution, each process computes f_{ij} functionality without blocking. P3 and P4 processes compute f_{12} and f_{22} respectively only after two events, e_1 and e_2, have been notified. These events denote that the inputs for f_{12} and for f_{22} functionalities, $a' = f_{11}(a)$ and $b'=f_{21}(b)$, are ready. In general, P3 and P4 have to handle a local status variable (not-represented in Fig.4) for registering the arrival of each event since e_1 and e_2 notifications could arrive in different deltas. Such handling is an additional functionality wrapping the original f_{i2} functionality, which results in a functionality f_{i2}', as shown in Fig.4.

The sketch in Fig. 4 enables several equivalent codes based on the fact that processes P_3 and P_4 can be written either as SC_METHOD processes with a static sensitivity list, or as SC_THREAD processes with an initial and unique wait statement (coded as a SystemC dynamic sensitivity list, but used as a static one), before the function computation. Moreover, as with the Fig. 3 cases, both in P_1 and in P_2, the execution of f_{i1} functionalities and event notifications can be swapped without repercussion on the fulfilment of equations (4-7).

Summarizing, the solutions shown are samples of the wide range of coding solutions for a simple specification problem. The richness of specification facilities and flexibility of SystemC enable each student to find at least one solution, and furthermore, to provide some different alternatives. However, such an open use of the language also leads to a variety of possible incorrect solutions. Fig. 5 illustrates only two of them.

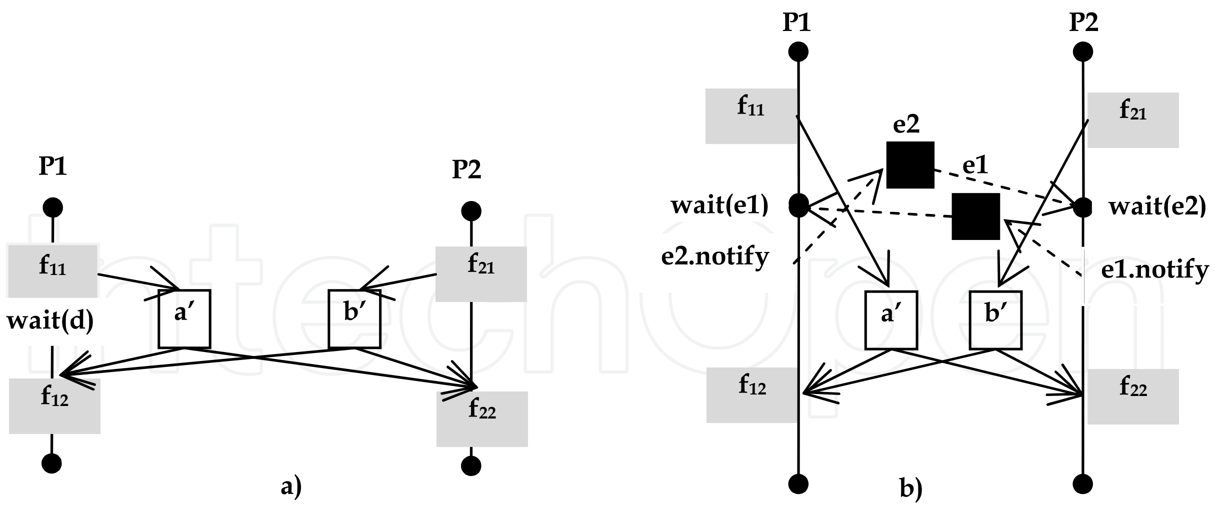


Fig. 5. Solution based on four finite and non-blocking processes.

In the Fig.5a example, the order condition (7) might be broken, and thus the specification intent in Fig.5a is not fulfilled. Under SystemC execution semantics, f_{22} may happen either before or after f_{11}. The former case can happen if P2 starts its execution first. SystemC is non-pre-emptive, thus f_{22} will execute immediately after f_{21}, and thus before the start of P1, which violates condition (7). Moreover, the example in Fig. 5a does not provide functional determinism because condition (7) might be fulfilled or not, which means that output z can present different output values for the same inputs. Therefore, it is not possible to make a deterministic prediction of what output z will be for the same set of inputs, since sometimes it can be $z=f_{22}(a,f_{21}(b))$, while others it can be $z=f_{22}(f_{11}(a),f_{21}(b))$. In many specification contexts functional determinism is required or at least desirable.

The Fig. 5b example shows another typical issue related to concurrency: deadlock. In Fig. 5b, a SystemC execution will always reach a point where both processes P_1 and P_2 get blocked forever, since the condition for them to reach the resumption can never be fulfilled. This is due to a circular dependency between their unblocking conditions. After reaching the wait statement, unblocking P_1 requires a notification on event e1. This notification will never come since P_2 is in turn waiting for a notification on event e2.

Even for the small parallel specification used in our experiment, al least one student was not able to find a correct solution. However, even for experienced designers it is not easy to validate and deal with concurrent specifications just by inspecting the code, relying and reasoning based on the execution semantics, even if they are supported by a graphical representation of the concurrency, synchronization and communication structure. Relatively small concurrent examples can present many alternatives for analysis. Things get worse with complex examples, where the user might need to compose blocks whose code is not known or even visible. Moreover, even simple concurrent codes, can present subtle bug conditions, which are hard to detect, but risky and likely to happen in the final implementation.

For example, let's consider a new solution of the 'simple' specification example based on the Fig.3a structure. It was already explained that this structure works well when considering either delta notification or timed notification. A user could be tempted to use immediate notification for speeding up the simulation with the Fig.3a structure. However, this specification would be non-deterministic. In effect, at the beginning of the simulation, both P1 and P2 are ready to execute in the first delta cycle. SystemC simulation semantics do not state which process should start in a valid simulation. If P1 starts, it will mean that the e2 immediate notification will get lost. This is because SystemC does not register immediate notification and requires the process receiving it (in this case P2) to be waiting for it already. Thus, there will be a partial deadlock in the specification. P2 will get blocked in the 'wait(e2)' statement forever and the output of P2 will be the null sequence z={}, while y={$f_{21}(f_{11}(a),f_{21}(b))$}. Assuming the functions of equations (3), for (a,b)=({1},{2}), (y,z) = ({6},{}). Symmetrically, if P2 starts the execution first, then P1 will get blocked forever at its wait statement, and the output will be y={}, z={$f_{22}(f_{11}(a),f_{21}(b))$}. Assuming the functions of equations (3), for (a,b)=({1},{2}), (y,z) = ({},{2}). Thus, in this case, no outputs correspond to the initial intention. There is functional non-determinism, and partial deadlock.

It is not recommended here that some properties should always be present (e.g., not every application requires functional determinism). Nor is the prohibition of some mechanisms for concurrent specification recommended. For instance, immediate notification was introduced in SystemC for SW modelling and can speed up simulation. Indeed, the Fig.3a example can deterministically use immediate notification with some modifications in the code for explicit registering of immediate events. However, such modification shows that the solution was not as straightforward as designers could initially think. Therefore, the definition of when and how to use such a construct is convenient in order to save wastage of time in debugging, or what it would be worse, a late detection of unexpected results.

Actually, what it is being stated is that concurrent specification becomes far from straightforward when the user wants to ensure that the specification avoids the plethora of issues which may easily appear in concurrent specifications (non-determinism, deadlock, starvation, etc), especially when the number of processes and their interrelations grow. Therefore, a first challenge which needs to be tackled is to provide methods or tools to detect that a specification can present any of the aforementioned issues. The following sections will introduce this problem in the context of SystemC simulation. The difficulty in being exhaustive with simulation-based techniques will be shown. Then the possibility to rely on correct by construction specification approaches will be discussed.

In order to simplify the discussion, the following sections will focus on functional determinism. In general, other issues, e.g. deadlock, are orthogonal to functional determinism. For instance, the Fig. 5b case presents deadlock while still being deterministic (whatever the input, each output is always the same, a null sequence). However, non-determinism is usually a source of other problems, since it usually leads to unexpected process states, for which the code was not prepared to avoid deadlock or other problems. Fig. 4a example with immediate notification was an example of this.

3. Simulation-based verification for flexible coding

Simulation-based verification requires the development of a verification environment. Fig. 6 represents a conventional SystemC verification environment. It includes a test bench, that is, a SystemC model of the actual environment where the system will be encrusted. The test bench is connected and compiled together with the SystemC description of the system as a single executable specification. When the OSCI SystemC library is used, the simulation kernel is also included in the executable specification. In order to simulate the model, the executable specification is launched. Then, the test bench provides the input stimuli to the system model, which produces the corresponding outputs. Those outputs are in turn collected and validated by the test bench.

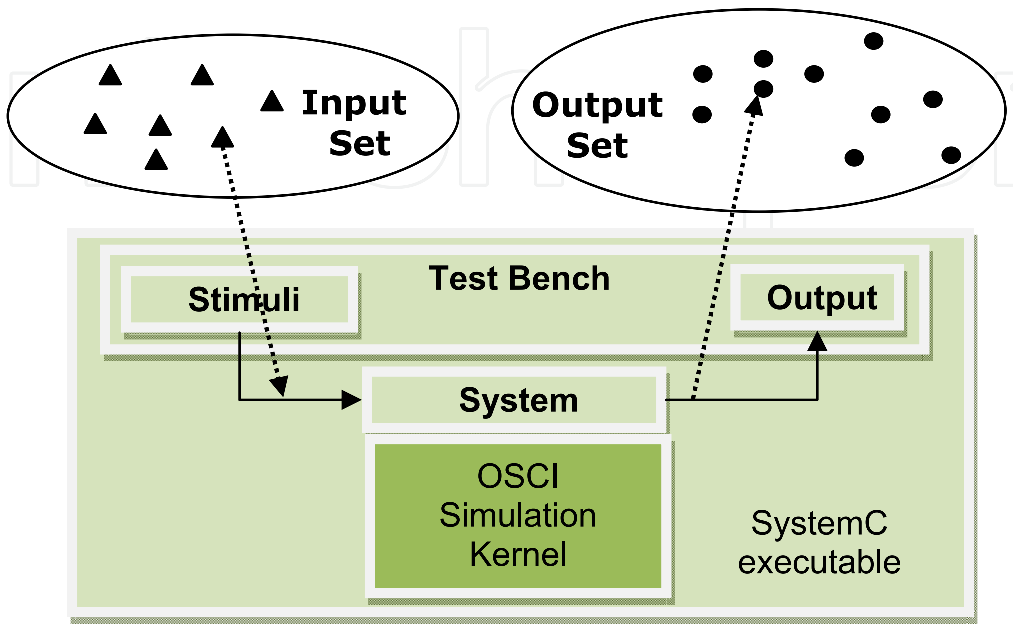


Fig. 6. Simulation-based verification environment with low coverage.

The Fig. 6 framework has a significant problem. A single execution of the executable specification provides very low verification coverage. This is due to two main factors:

- The test bench only reflects a subset of the whole set of possible inputs which can be fed by the actual environment (Input Set).
- Concurrency implies that, for each fixed input (triangle in Fig. 6), there are in general more than one feasible execution order or *scheduling*, thus potentially, more than one feasible output. However, a single simulation shows only one *scheduling*.

The first point will be addressed in section 3.1. The following sections will focus on dealing with how to tackle verification when concurrency appears in the specification.

3.1 Stimuli generation

Assuming a fully sequential system specification, the first problem consists in finding a sufficient number of stimuli for a 'satisfactory' verification of the specification code. Satisfactory can mean 100% or a sufficiently high percentage of a specific coverage metric.

Therefore, an important question is which coverage metrics to use. A typical coverage metric is branch coverage, but there are more code coverage metrics, such as lines, blocks, branches, expressions, paths, and boundary-path. Other techniques (Fallah, 1998); (Gupta, 2002); (Ugarte, 2011) are based on functional coverage metrics. Functional coverage metrics are defined by the engineer, and thus rely on engineer experience. They can provide better performance in bug detection than code coverage metrics. However, code coverage metrics do not depend on the engineer, thus they can be more easily automated. They are also simpler, and provide a first quality metric of the input set.

In complex cases, an exhaustive generation of input vectors is not feasible. Then, the question is which vectors to generate and how to generate them. A basic solution is random generation of input vectors, (Kuo, 2007). The advantages are simplicity, fast execution speed and many uncovered bugs with the first stimulus. However, the main disadvantages are twofold: first, many sets of input values might lead to the same observable behaviour and are thus redundant, and second, the probability of selecting particular inputs corresponding to corner cases causing buggy behaviour may be very small.

An alternative to random generation is, constrained random vector generation, (Yuan, 2004). Environments enabling constrained random generation enable a random, but controlled generation of input vectors by imposing some bounds (constraints) on the input data. This enables a generation of input vectors that are more representative of the expected environment. For instance, one can generate values for an address bus in a certain range of the memory map. Constrained randomization also enables a more efficient generation of input vectors, once they can be better directed to reach parts of code that a simple random generation will either be unlikely to reach or will reach at the cost of a huge number of input stimuli. In the SystemC context, the SystemC Verification library (SCV) (OSCI, 2003), is an open source freely available library which provides facilities for constrained randomization of input vectors. Moreover, the SCV library provides facilities for controlling the statistical profile in the vector generation. That is, the user can apply typical distribution functions, and even define customized distribution functions, for the stimuli generated. There are also commercial versions such as Incisive Specman Cadence (Kuhn, 2001), VCS of Synopsys, and Questa Advanced Simulator of Mentor Graphics. The inconvenience of constrained random generation of input vectors is the effort required to generate the constraints. It already requires extracting information from the specification, and relies on the experience of the engineer. Moreover, there is a significant increase in the computational effort required for the generation of vectors, which needs solvers.

More recently, techniques for automatic generation of input vectors have been proposed (Godefroid, 2005); (Sen, 2005); (Cadar, 2008). These techniques use a coverage metric to guide (or direct) the generation of vectors, and bound the amount of vectors generated as a function of a certain target coverage. However, these techniques for automatic vector generation require constrained usage of the specification language, which limits the complexity of the description that they can handle.

In order to explain these strategies, we will use an example consisting in a sequential specification which executes the f_{ij} functionalities in Fig. 1 in the following order {f_{11}, f_{21}, f_{12}, f_{22}}. Therefore, this is an execution sequence fulfilling the specification intent, provided the dependency graph in Fig. 1b. Let's assume that the specific functions of this sequential system are given by equations (3), and that the metric to guide the vector generation is branch coverage. It will also be assumed that the inputs ('a' and 'b') are of integer type with range [-2,147,483,648 to 2,147,483,647]. A first observation to make is that our example will have two execution paths, defined by the control statements, specifically, the conditional function f_{22}. Entering one or another path depends on the value of the 'x_1' input of f_{22}, which in turn depends on the input to f_{11}, that is, on the input 'a'.

By following the first strategy, namely, running the executable specification with random vectors of 'a' and 'b', it will be unlikely to reach the true branch of the control sentence within f_{22}, since the probability of reaching it is less than 2.5E-10 for each input vector. Even if we provide means to avoid repeating an input vector, we could need 2.5E10 simulations to reach the true path.

Under the second strategy, the verification engineer has to define a constraint to increase the probability of reaching the true branch. In this simple example, the constraint could be the creation of a weighted distribution for the x input, so that some values are chosen more often than others. For instance, the following sentence: *dist {[min_value:25713]:= 33, 25714:= 34, [25715:max_value]:=33}*, states that the value that reaches the true branch of f_{22}, that is, 25,714, has a 33.3% probability to be produced by the random generator. The likelihood of generation of values below 25.714 would be 33.3%, and similarly 33.3% for values over 25,714. Thus, the average number of vectors required for covering the two paths would be 3. Then, the user could prepare the environment for producing three input vectors (or a slightly bigger number of them for safety). One possible vector set generated could be: (a,b) = {(12390, -2344), (-3949, 1234), (25714, -34959)}. The efficiency of this method relies on the user experience. Specifically, the user has to know or guess which values can lead to different execution paths, and thus which groups of input values will likely involve different behaviours.

The latter strategy would be directed vector generation. This strategy analyses the code in order to generate the minimum set of vectors for covering all branches. Directing the generation in order to cover all execution paths would be the ideal goal. However, this makes the problem explode. In the simple case in Fig. 1, branch and path coverage is the same since there is only one control statement. In this case, only one vector is required per branch. For example, the first value generated could be random, e.g., (a = 39349, b= -1024). As a result, the system executes the false path of the control statement. The constraint of the executed path is detected and the constraint of the other branch generated. In this case, the constraint is a=25714. The generator solves the constraint and produces the next vector (a, b) = (25714, 203405). With this vector, the branch coverage reaches 100% of coverage and vector generation finishes. Therefore, the stimulus set is (a,b) = { (39349, 1024), (25714, 203405)}.

3.2 Introducing concurrency: scheduling coverage

In the previous section, the generation of input vectors for reaching certain coverage (usually of branches or of execution paths) has been discussed. For this, we assumed a sequential specification, which means that for a fixed input vector, a fixed output vector is expected. Thus, the work focuses on finding vectors for exercising the different paths which can be executed by the real code, since these paths reflect the different behaviours that the code can exhibit for each input. Each type of behaviour is a relationship between the input and the output. Functional behaviour will imply a single output for given input.

As was mentioned at the beginning of section 3, the injection of concurrency in the specification raises a second issue. Concurrency makes it necessary to consider the possibility of several schedulings for the execution of the system functionality for a fixed input vector. This can potentially lead to different behaviours for the same input. At specification level, there are no design decisions imposing timing and thus no strict ordering

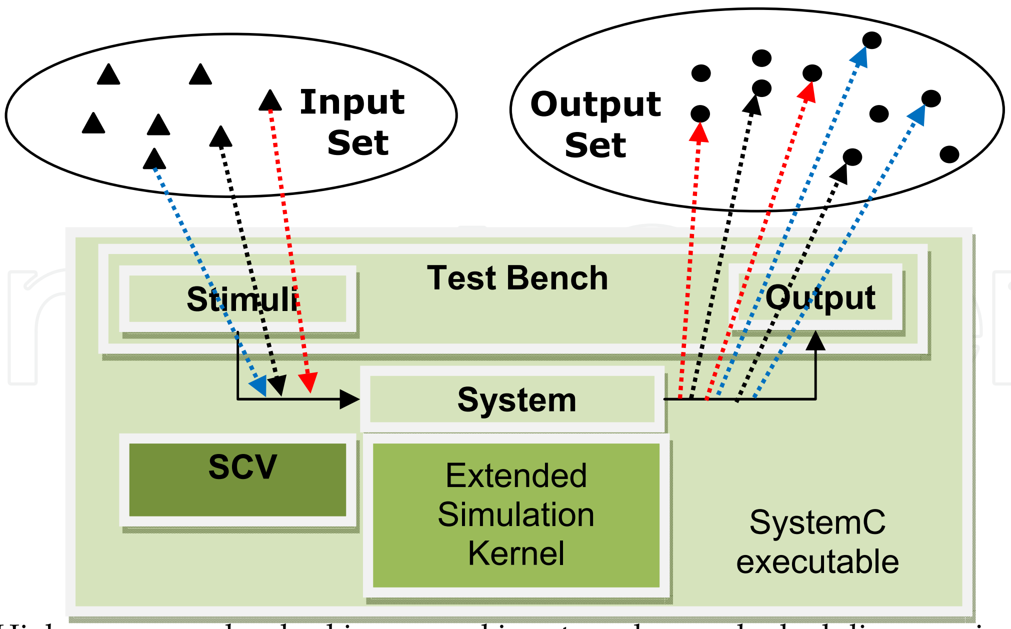


Fig. 7. Higher coverage by checking several inputs and several schedulings per input.

to the computation of the concurrent functionality, thus all feasible order must be taken into account. The only exception is the timing of the environment, which can be neglected for generality. In other words, inputs can be considered as arriving in any order.

In order to tackle this issue, Fig. 7 shows the verification environment based on multiple simulations proposed by (Herrera, 2006). Using multiple simulations, that is, multiple executions (ME) in a SystemC-based framework, enables the possibility of feeding different input combinations. SystemC LRM comprises the possibility of launching several simulations from the same executable specification through several calls to the *sc_elab_and_sim* function. (Herrera, 2006), and (Herrera, 2009), explain how this could be done in SystemC. However, SystemC LRM also states that such support depends on the implementation of the SystemC simulator. Currently, the OSCI simulator does not support this feature. Thus, it can be assumed that running N_E simulations currently means running the SystemC executable specification N_E times. In (Herrera, 2006), and (Herrera, 2009), the launch of several simulations is automated through an independent launcher application.

The problem is how to simulate different scheduling, and thus potentially different behaviour, for each single input. Initially, one can try to perform several simulations for a fixed input test bench (one triangle in the Fig. 7 schema,). However, by using the OSCI SystemC simulator, and most of the available SystemC simulators, only one scheduling is simulated. In order to demonstrate the problem, we define a scheduling as a sequence of segments (s_{ij}). A scheduling reflects a possible execution order of segments under SystemC semantics. A segment is a piece of code executed without any pre-emption between calls to the SystemC scheduler, which can then make a scheduling decision (SDi). A segment is usually delimited by blocking statements. A scheduling can be characterized by a specific sequence of scheduling decisions. In turn, the set of feasible schedulings of a specification can be represented in a compact way through a scheduling decision tree (SDT). For instance, Fig. 8 shows the SDT of the Fig. 2 (and Fig. 3) specification. This SDT shows that there are 4 possible schedulings (S_i in Fig. 8). Each segment is represented as a line ended with a black

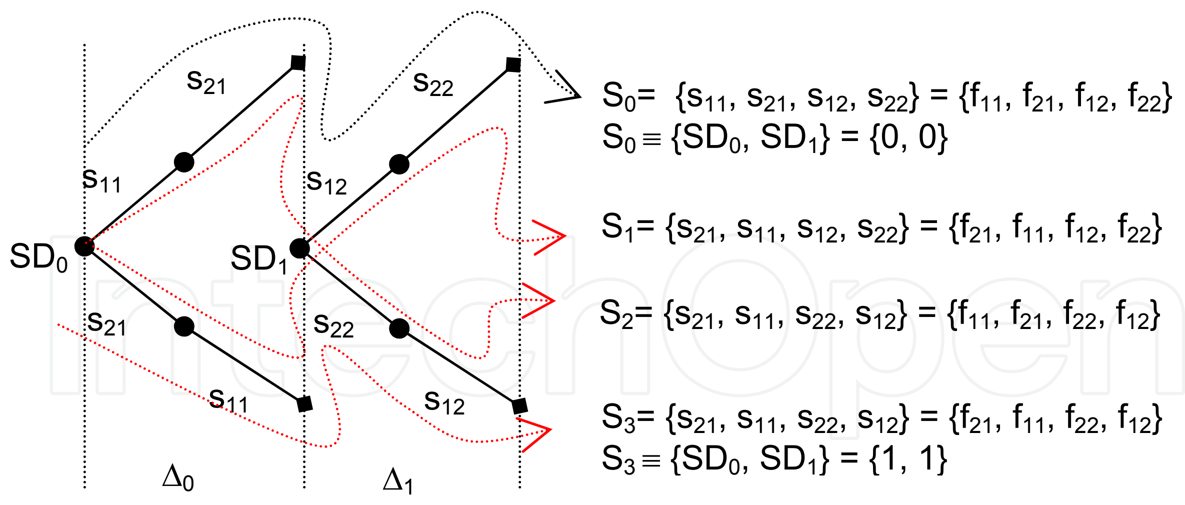

s_{21}
s_{22}
s_{11}
s_{12}
SD_0
SD_1
s_{21}
s_{22}
s_{11}
s_{12}
Δ_0
Δ_1
$S_0 = \{s_{11}, s_{21}, s_{12}, s_{22}\} = \{f_{11}, f_{21}, f_{12}, f_{22}\}$
$S_0 \equiv \{SD_0, SD_1\} = \{0, 0\}$
$S_1 = \{s_{21}, s_{11}, s_{12}, s_{22}\} = \{f_{21}, f_{11}, f_{12}, f_{22}\}$
$S_2 = \{s_{21}, s_{11}, s_{22}, s_{12}\} = \{f_{11}, f_{21}, f_{22}, f_{12}\}$
$S_3 = \{s_{21}, s_{11}, s_{22}, s_{12}\} = \{f_{21}, f_{11}, f_{22}, f_{12}\}$
$S_3 \equiv \{SD_0, SD_1\} = \{1, 1\}$

Fig. 8. Scheduling Decision Tree for the examples in Fig. 2 and Fig. 3.

dot. Moreover, in the Fig. 8 example, each s_{ij} segment corresponds to a f_{ij} functionality, computed in this execution segment. Each dot in Fig. 8 reflects a call to the SystemC scheduler. Therefore, each simulation of the Fig. 2, and Fig. 3 examples, either with delta or timed notification, always involves 4 calls to the SystemC scheduler after simulation starts. However, only two of them require an actual selection among two or more processes ready to execute, that is, a scheduling decision (SD_i). As was mentioned, multiple executions of the executable simulation compiled against the existing simulators would exhibit only a single scheduling, for instance S_0 in the Fig. 8 example. Therefore, the remaining schedulings, S_1, S_2 and S_3 would never be checked, no matter how many times the simulation is launched.

As was explained in section 2, the Fig. 2 and Fig. 3 examples fulfil the partial order defined by equations (4-7), so the unchecked schedulings will produce the same result. This is easy to deduce by considering that each segment corresponds to a f_{ij} functionality of the example.

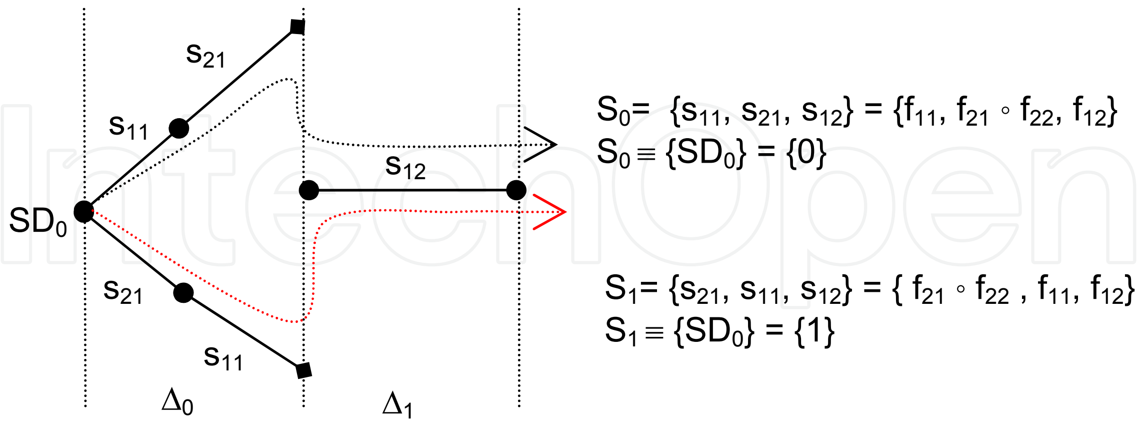

s_{21}
s_{11}
s_{12}
SD_0
s_{21}
s_{11}
Δ_0
Δ_1
$S_0 = \{s_{11}, s_{21}, s_{12}\} = \{f_{11}, f_{21} \circ f_{22}, f_{12}\}$
$S_0 \equiv \{SD_0\} = \{0\}$
$S_1 = \{s_{21}, s_{11}, s_{12}\} = \{ f_{21} \circ f_{22} , f_{11}, f_{12}\}$
$S_1 \equiv \{SD_0\} = \{1\}$

Fig. 9. Scheduling Decision Tree for the Fig.2 and Fig. 3 examples.

However, let's consider the Scheduling Decision Tree (SDT) in the Fig. 5a example, shown in Fig. 9. The lack of a wait statement between f_{21} and f_{22} in P2 in the Fig. 5a example implies that P_2 executes all its functionality (f_{21} and f_{22}) in a single segment (s_{21}). Notice that a segment can comprise different functionalities, or, as in this case, one functionality as a result of composition of f_{21} and f_{22} (denoted $f_{21} \circ f_{22}$). Therefore, for the Fig. 5a example, the SystemC kernel executes three segments, instead of four as in the case of Fig. 4 example. Notice also that several scheduler calls can appear within the boundaries of a delta cycle.

The SDT of the Fig. 5 example has only a single scheduling decision. Therefore, two schedulings are feasible, denoted S_0 and S_1. However, only one of them, S_0, fulfils the partial order defined by equations (4-7). As was mentioned, the OSCI simulator will execute only one, either S_0 or S_1, even if we run the simulation several times. This is due to practical reasons, since OSCI and other SystemC simulators implement a fast and straightforward scheduling based on a first-in first-out (FIFO) policy. If we are lucky, S_1 will be executed, and we will establish that there is a bug in our concurrent specification. However, if we are not lucky, and S_0 is always executed, then the bug will never be apparent. Thus, we can get the false impression of facing a deterministic concurrent specification.

Therefore, a simulation-based environment requires some capability for observing the different schedulings, ideally 100% coverage of schedulings, which are feasible for a fixed input. Current OSCI implementation of the SystemC simulation kernel fulfils the SystemC semantics and enables fast scheduling decisions. However, it produces a deterministic sequence of scheduling decisions, which is not changed from simulation to simulation for a fixed input. This has leveraged several techniques for enabling an improvement of the scheduling coverage. Before introducing them, a set of metrics for comparing different techniques for improving scheduling coverage of simulation-based verification techniques, proposed in (Herrera, 2006), will be introduced. They can be used for a more formal comparison of the techniques discussed here. These metrics are dependent on each input vector, calculated by means of any of the techniques explained in section 3.1.

Let's denote the whole set of schedulings S, where S = $\{S_0, S_1, \ldots, S_{size(s)}\}$, and size(S) is the total number of feasible schedulings for a fixed input. Then, the Scheduling Coverage, C_S, is the number of checked schedulings with regard to the total number of possible schedulings.

$$C_S = \frac{N_S}{size(S)} \tag{8}$$

The Multiple Execution Efficiency η_{ME} is the actual number of (non-repeated) schedulings N_S covered after N_E simulations (executions in SystemC).

$$\eta_{ME} = \frac{N_S}{N_E} = \frac{N_S}{N_S + N_R} = \frac{1}{1 + R_E} \tag{9}$$

N_R stands for the amount of repeated schedulings, which are not useful. As can be seen, η_{ME} can be expressed in terms of R_S. R_S is a factor which accounts for the number of repeated schedulings out of the total number of simulations N_E.

The total number of simulations to be performed to reach a specific scheduling coverage, $N_T(C_S)$ can be expressed as a function of the desired coverage, the number of possible schedulings, and the multiple execution efficiency.

$$N_T(C_S) = \frac{C_S \cdot size(S)}{\eta_{ME}} \tag{10}$$

Finally, the Time Cost for achieving a coverage C_S is approximated by the following equation:

$$T_E \approx \frac{C(TE)\cdot size(TE)}{\eta_{ME}}\cdot \bar{t} \tag{11}$$

Where $\bar{t}$ is the average simulation time of each scheduling. It is actually a rough approximation, since each scheduling can derive in shorter or longer schedulings. It also depends on the actual scheduling technique. However, equations (8-11) will be sufficiently useful for comparing the techniques introduced in the following sections, and the yield of conventional SystemC simulators, including the OSCI SystemC library in the simulation-based verification environments shown in Fig. 7. Conventional SystemC simulators provide a very limited scheduling coverage, $C_S = \frac{1}{size(S)}$, since N_S=1. Moreover, the scheduling coverage is fixed and cannot grow with further simulations. Since size(S) exponentially grows when adding tasks and synchronization mechanisms, the scheduling coverage quickly becomes low even with small examples. For instance, in (Herrera, 2006), a simple extension of the Fig. 2 example to three processes, each of three segments, leads to size(S)=216, thus C_S=0.46%.

3.2.1 Random and pseudo-random scheduling

The user of an OSCI simulator can try a trick to check different schedulings in a SystemC specification. It consists in changing the order of declaration of SystemC processes in the module constructor. Thus, the result of the first dispatching of the OSCI simulator at the beginning of the simulation can be changed. However, this trick gives no control over further scheduling decisions. Moreover, checking a different scheduling requires the modification of the specification code.

A simple alternative for getting multiple executions to exhibit different schedulings is changing the simulation kernel to enable a random selection among the processes ready to execute in each scheduling decision. Random scheduling enables $\frac{1}{size(S)} \leq C_S \leq 1$, and a monotonic growth of C_s with the number of simulations N_E. The dispatching is still fast, since it only requires the random generation of an index suitable for the number of processes ready to execute in each scheduling decision. The implementation can range from more complex ones guaranteeing the equal likelihood in the selection of each process in the ready-to-execute list, to simpler ones, such as the one proposed in (Herrera, 2006), which is faster and has low impact in the equal likelihood of the selection.

There are still better alternatives to pure random scheduling. In (Herrera, 2006), pseudorandom (PR) scheduling is proposed. Pseudorandom scheduling consists in enabling a pseudo-random, but deterministic, sequence of scheduling decisions from an initial seed. This provides the advantage of making each scheduling reproducible in a further execution. This reproducibility is important since it enables to debug the system with the scheduling which showed an issue (unexpected result, deadlock, etc) as many times as desired. Without this reproducibility, the simulation-based verification framework would be able to detect there is an issue, but would not be practically applicable for debugging it. Therefore, Pseudorandom scheduling presents the same coverage, $\frac{1}{size(S)} \le C_S \le 1$, and monotonic growth as C_S with the number of simulations of pure random scheduling. A freely available extension of the OSCI kernel, which implements and makes available Pseudorandom scheduling (for SC_THREAD processes) is provided in (UCSCKext, 2011).

Pseudorandom scheduling still presents issues. One issue is that, despite the monotonic growth of C_S with N_E, this growth is approximately logarithmic, due to the probability of finding a new scheduling with the number of simulations performed. Each new scheduling found reduces the number of new schedulings to be found, and Pseudorandom schedulings have no mechanisms to direct the search of new schedulings. Thus, in pseudorandom scheduling, $\eta_{ME} \le 1$ in general, and it quickly tends to 0 when N_E grows. Another issue is that it does not provide specification-independent criteria to know when a specific C_S or a size(S) has been reached. C_S or size(S) can be guessed for some concurrency structures.

3.2.2 Exhaustive scheduling

In (Herrera, 2009), a technique for directing scheduling decisions for an efficient and exhaustive coverage of schedulings, called DEC scheduling, was proposed. The basic idea, was to direct scheduling decisions in such a way that the sequence of simulations perform a depth-first search (DFS) of the SDT. For an efficient implementation, (Herrera, 2009), proposes to use a scheduling decision register (SDR), which stores the sequence of decisions taken in the last simulation.

For instance, for the Fig. 8 SDT, corresponding to examples in Fig.2 and 3, the first simulation will produce the S_0 scheduling. This means that the SDR will be $SDR_0=\{0,0\}$, matching the FIFO scheduling semantics of conventional SystemC simulators, where the first process in the ready-to-execute queue is always selected. Then, a second simulation under the DEC scheduling, will use the SDR to reproduce the scheduling sequence until the penultimate decision (also included). Then, the last decision is changed. Remember that a scheduling decision SD_i is taken whenever a selection among at least two ready-to-execute processes is required. Since in the previous simulation the last scheduling decision was to select the 0-th process (denoted in the example as $SD_1=0$), in the current simulation the next process available in the ready-to-execute queue is selected (that is, $SD_1=1$). Therefore, the second execution in the example simulates the next scheduling of the SDT, $S_1=\{0,1\}$.

In a general case, the change in the selection of the last decision can mean an extension of the SDT (which means that the simulation must go on, and so go deeper into the SDT). Another possibility is what happens in the example shown, where the branch at the current depth level has been fully explored and a back trace is required. In our example, the third simulation will go back to SD_0 decision and will look for a different scheduling decision ($SD_0=1$). What will occur in this case is that the simulation can go on and new scheduling decisions, will be required, thus requiring the extension of the SDR again, and thus leading to the $S_2=\{1,0\}$ scheduling. Following the same reasoning, it is straightforward to deduce that the next simulation will produce the scheduling $S_3=\{1,0\}$.

Therefore, the main advantage of DEC scheduling with regard to PR scheduling is that $\eta_{ME} = 1$. That is, each new simulation guarantees the exploration of a new scheduling. This provides a more efficient search since the scheduling coverage grows linearly with the number of simulations. That is, for DEC scheduling:

$$\frac{1}{size(S)} \le C_S = \frac{N_E}{size(S)} \le 1 \qquad (12)$$

Another advantage of DEC scheduling is that it provides criteria for finishing the exploration of schedulings which does not require an analysis of the specification. It is possible thanks to the ordered exploration of the SDT, (Herrera, 2009). The condition for finishing the exploration is fulfilled once a simulation (indeed the N_E=size(S)-th simulation) has selected the last available process for each scheduling decision of the SDR, and no SDT extension (that is, no further events and longer simulation) is required. In the example in Fig. 8, this corresponds to the scheduling S_3={1,1}. When this condition is fulfilled, 100% scheduling coverage (C_S) has been reached. Notice that, in order to check the fulfilment of the condition, no estimation of size(S) is necessary, thus no analysis of the concurrency and synchronization structure of the specification is required. In the case that size(S) can be calculated, e.g. because the concurrency and synchronization structure of the specification is regular or sufficiently simple, then C_S, can be calculated through equation (12). For instance, in the Fig. 8 example size(S)=4, then, applying equation (8), C_S=0.25N_S.

The main limitation of DEC scheduling is that size(S) has an exponentially growth for a linear growth of concurrency. Thus, although $\eta_{ME} = 1$ is fulfilled, the specification will exhibit a state explosion problem. The state explosion problem is exemplified in (Godefroid, 1995), which shows how a simple philosopher's example can pass from 10 states to almost 10^6 states when the number of philosophers grows from two up to twelve. Another related downside is that a long SDR has to be stored in hard disk, thus the reproduction of scheduling decisions will include the time penalties for accessing the file system. This means a growth of $\bar{t}$ in equation (11) for the calculation of the simulation-based verification time, which has to be taken into account when comparing DEC scheduling with Pseudo-random or pure random techniques, where scheduling decisions are lighter.

3.3 Partial Order Reduction techniques

A set of simulation-based techniques, based on Partial Order Reduction (POR) has been proposed for tackling the state explosion problem. POR is a partition-based testing technique, based on the execution of a single representative scheduling for each class of equivalent schedulings. This reduces the number of schedulings to be explored, from size(S) feasible schedulings, to M, with M<size(S). M is the number of sets of non-equivalent scheduling classes, each one enclosing a set of equivalent schedulings. The equivalence is understood in functional terms. That is, the simulation of two schedulings of an equivalent scheduling class will lead to the same state, and therefore to the same effect on the system behaviour. When applying POR techniques, the objective is not to achieve C_S=100%, but C_M=100%, where C_M stands for the coverage of representative (non-equivalent) schedulings. Expressed in other terms, a single simulation serves to check on average a set of $\overline{L}$ equivalent simulations. Thus POR techniques enable a scheduling

coverage of $\frac{N_E \cdot \bar{L}}{size(S)}$ and efficiencies greater than 1, that is, $\eta_{ME} = \frac{N_S}{N_E} \geq 1$. Obviously, the efficiency in the exploration of non-equivalent schedulings will always remain below or equal to 1.

In order to deduce which schedulings are equivalent, POR methods require the extraction and analysis of information from the specification, in order to study when the possible interactions and dependencies between processes may lead or not to functionally equivalent paths. For instance, the detection of shared variables, and the analysis of write-after-write, read-after-write, and write-after-read situations in them, enable the extraction of non-equivalent paths which can lead to race conditions. Similarly, event synchronization has to be analyzed (notification after wait, wait after notification, etc) since non-persistence of events can lead to misses and to unexpected deadlock situations, non-determinism or other undesirable effects. (Helmstetter, 2006) and (Helmstetter, 2007) propose dynamic POR (DPOR) of SystemC models, by adapting dynamic POR techniques initially developed for software (Flanagan, 2005). Dynamic POR selects the paths to be checked during the simulation, in each scheduling decision, performing the analysis among ready-to-execute processes. Later works, such as the 'Satya' framework (Kundu, 2008), have proposed the combination of static POR techniques with dynamic POR techniques. The basic idea is that the runtime overhead is reduced by computing the dependency information statically; to later use it during runtime.

As an example, let's consider the first scheduling decision (SD_0) in the SDT in Fig. 8 for any of the specifications represented by Fig. 2 and 3. Depending on SD_0, the scheduling executed can start either by $\{s_{11}, s_{21}, \ldots\}$ or by $\{s_{21}, s_{11}, \ldots\}$, each one representing two different classes of schedulings, $\{S_0, S_1\}$ and $\{S_2, S_3\}$ respectively. A POR analysis focused on the impact on functionality, will establish that those scheduling classes actually account for the following two possible starting sequences in functional terms, either $\{f_{11}, f_{21}, \ldots\}$ or $\{f_{21}, f_{11}, \ldots\}$. A POR technique will establish that f_{11} and f_{21} have impact on some intermediate and shared variables, 'a' and 'b', which reflect the state of the concurrent system and which imply dependencies between P_1 and P_2, thus requiring a specific analysis. Specifically, the POR technique will establish that those two possible initializations of the schedulings lead to the same state (in the next delta, Δ_1), described by $a'=f_{11}(a)$ and $b'=f_{11}(b)$. In other words, since there are no dependencies, any starting sequence leads to the same intermediate state, and schedulings starting with $SD_0=0$, that is, starting by $\{s_{11}, s_{21}, \ldots\}$, and schedulings starting with $SD_0=1$, that is, starting by $\{s_{21}, s_{11}, \ldots\}$ will be equivalent if they keep the same sequence of decisions in the rest of the sequence of scheduling decisions (SD_0). Therefore only one of the alternatives in SD_0 has to be explored. This idea can be iteratively applied generally leading to a drastic reduction in the number of paths which have to be explored, thus fulfilling $M<<size(s)$. Such a drastic reduction can be observed in our simple example if we continue with it. Let's take, for instance, $SD_0=0$ in the example, and let's continue the application of a dynamic POR. At this stage, in the worst case, we will need to execute S_0 and S_1, thus $M=2$ simulations for a complete coverage of functional equivalent schedulings. Furthermore, DPOR is again applied for the second delta, Δ_1. Considering y and z as state variables directly forwarded to the outputs, there is no read after write, write after read or write after write dependency among them. Therefore, it can be concluded that the decision on SD_1 will be irrelevant in reaching the same (y, z) state after the Δ_1 delta. Therefore, $M=1$,

and $\eta_{ME} = 4$ in this case, since any of the four schedulings exposed by a single simulation will be representative of a single class of schedulings, equivalent in functional terms.

The method described in (Helmstetter, 2006) is complete, but not minimal, since it is feasible to think about specifications where M non-equivalent schedulings lead to different states, but where those different states are not translated into different outputs. This means that M would still admit a further reduction. This reduction would require an additional analysis of the actual relationship between state variables and the outputs. As an example, let's consider that in our examples in Fig. 2, z was not considered as a system output, but as informative or debugging data, resulting from post-processing, through f_{22}, an internal state variable), and that the only output is y. Thus, it would demonstrate the irrelevance of the SD_1 scheduling decision, which would save the last DPOR analysis in Δ_1.

The approach of (Helmstetter, 2006) is also fork-based. Whenever a scheduling decision finds non-equivalent or potentially non-equivalent paths, the simulation is spawned in order to enable a concurrent check. Thus, several non-equivalent groups of schedulings can be explored by launching a single simulation. This makes η_{ME} even bigger, and $\eta_{ME} = N_S \geq 1$, up to the point where a single simulation could cover all the scheduling classes. However, this optimization should be carefully considered. In order to give an actual speed up to the verification, it is necessary that the simulation engine can take advantage of a multi-core host machine. In (Helmstetter, 2006), the first advances for a parallel SystemC simulator are given. If the simulation is sequential, then a fork-based approach can easily be counter-productive in terms of time cost even if SystemC simulators with actual parallel simulation capabilities are available.

In general, the main limitation of POR-based approaches is their need for extracting information from the specification. The limitations of the front-end tools used for extracting the information used for static dependency analysis, and the need to make the analysis feasible limit the supported input code. Specifically, the approach of (Helmstetter, 2006) is restricted to the SystemC subset admitted by the open-source and freely available Pinapa front-end (Moy, 2005). Satya is based in the commercial EDG C++ front-end, which provides wider support than Pinapa. However, it still presents limitations for supporting features such as dynamic casting and process creation. The work of (Sen, 2008) claims its independency from any external parser, while being able to detect potential errors in an observed execution, even if the error does not take place in the actual simulation. However, its goal is temporal assertion-based verification, rather than improving test coverage.

3.4 Merging scheduling techniques

In (Herrera, 09), the local application and cooperation of different scheduling techniques (PR, DEC and POR) is proposed. Two types of localities are distinguished:

- Spatial Locality: in order to improve scheduling coverage for a specific group of processes of the system specification.
- Temporal Locality: in order to improve scheduling coverage in a specific interval of the simulation time.

For instance, in some parts of the specification where SystemC is used in a flexible manner, e.g., a high-level concurrent model of an intellectual property (IP) block, DEC scheduling could be applied. Then POR could be applied to other parts, e.g., an in-house TLM platform, where the IP block is connected, and whose code can be bound to the specification rules stated by the POR technique. Table 1 summarizes the main characteristics of the different scheduling techniques reviewed.

Scheduling Technique	C_S	η_{ME}	Reproducibility	Linear growth of C_S with N_E	Specification Independent Detection of C_S=1	Specification Analysis Required
FIFO (OSCI simulator)	$\frac{1}{size(S)}$	$\frac{1}{N_E}$	yes	no	no	no
Random	$\geq \frac{1}{size(S)}$ ≤ 1	$\geq \frac{1}{N_E}$ ≤ 1	no	no	no	no
Pseudo Random	$\geq \frac{1}{size(S)}$ ≤ 1	$\geq \frac{1}{N_E}$ ≤ 1	yes	no	no	no
DEC	$\frac{N_E}{size(S)}$	1	yes	yes	yes	no
POR	$\frac{N_E \cdot \bar{L}}{size(S)}$	≥ 1 $\leq L$	yes	yes	yes	yes

Table 1. Comparison of scheduling techniques for simulation-based verification.

4. Methodologies for early correct specification

As shown in the previous sections, the success of a simulation-based verification methodology greatly depends on the ability to explore the effects of all the feasible execution alternatives, or at least, the “equivalent ones”. The problem is already challenging for sequential specifications, especially for control-oriented algorithms, and becomes practically intractable when concurrency appears in the specification, since the number of execution paths grows exponentially.

As has been shown, a way to tackle the explosion problem, for finding both a more reduced and efficient set of input vector generation, and an efficient set of schedulings, is the usage of information from the specification. Automated test generation techniques direct vector generation by detecting control statements and looking for vectors which exercise their different branches. Similarly, partial order reduction techniques need to analyze, either statically or dynamically, which variables or events produce dependencies among processes in order to extract the representative schedulings which need to be simulated.

This means that some conditions for making the specification wrong and hard to verify are already known. Thus, a different perspective is possible. Why not build specification methodologies which oblige, or at least help, the user to avoid such source problems, instead of letting them appear in the specification, with the consequential requirement of a costly verification.

An alternative consists in building specification methodologies which selectively adopt certain specification rules. Such rules will enable enough expressivity to solve the specification problem, but at the same time they rely on formal conditions for building correct specifications. By assuming the fulfilment of such specification rules, the formal support ensures the fulfilment of the properties pursued, or at least enables the application of analysis techniques for assessing such fulfilment. This idea is generally applicable. For instance, a methodology could forbid the usage of control statements. Then the specification would have just one data path, and the generation of test input vectors would be drastically simplified. However, this type of coding constraint would be very restrictive in many application domains, where user needs control sentences. Each specification methodology has its expressivity requirements, which puts bounds on the specification rules.

Embedded system specification requires expressing concurrency and abstraction. It might easily lead to the SystemC user to run into the plethora of issues associated to concurrency (non-determinism, deadlock, starvation, etc), as was illustrated in section 3. However, if certain smart rules are imposed on how concurrency is expressed in SystemC, it can highly facilitate to build early correct concurrent specifications. This principle has inspired several works, such as SystemC-H (Patel, 2004), SysteMoC (Haubelt, 2007), HetSC (Herrera, 2007), and HetMoC (Zhu, 2010), which have proposed SystemC specification methodologies to ensure, or facilitate the verification, of certain properties. These methodologies state a set of SystemC facilities (and provide additional ones when they are not provided by the standard core of SystemC) and state a set of specification rules. Methodologies such as HetSC, SystemC-H and SysteMoC rely on well-known formalisms, related to specific Models of Computation (MoC), such as Khan Process Networks (KPN) (Kahn, 1974), Synchronous Data Flows (SDF) (Lee, 1987), Concurrent Sequential Processes (CSP), Synchronous Reactive (SR) systems, and Dynamic Data Flows (DDF). HetMoC, relies on the ForSyDe formalism (Jantsch, 2004), which targets the unification of several MoCs. Finally, a standard extension of the SystemC language, such as SystemC-AMS, adopts a variation of the SDF MoC, called T-SDF, which annotates a time advance after each cluster execution.

Two important factors which characterize these types of specification methodologies are the properties targeted and the way these are achieved, that is, which specification facilities, specification rules, and assumptions configure the methodology. Two typical properties pursued are functional determinism and deadlock protection. A relatively flexible way to ensure functional determinisms is to build the specification methodology according to the KPN formalism. The adoption of a more constrained specification style, through a specification methodology which fulfils the SDF formalism, enables the application of an analysis for ensuring deadlock protection, as well as functional determinism. This is illustrated through the Fig. 10 example.

Fig. 10a shows the structure of a HetSC specification for solving the Fig.1 specification problem. HetSC states the rules to be followed in the SystemC coding for building the concurrent solution as a Khan Process Network. There are rules regarding the facilities to use (SC_THREADS for P1 and P2, and blocking fifo channels with infinite buffering capability, that is, channels of uc_inf_fifo type, provided by the HetSC library). There are rules regarding how to write the processes, e.g., only one channel instance can be accessed (either for reading or for writing) at a time. Finally, there are rules regarding communication and computation, e.g., no more than one process can access a channel instance either as a reader or as a writer. More details on the rules can be found at the (HetSC website, 2012). All these SystemC coding rules are designed to fulfil the rules and assumptions stated in Kahn, 1974. Provided they are fulfilled, as happens in the Fig. 10a case, it can be said that the Fig.10a specification is functionally deterministic. Notice that read accesses to the uc_inf_fifo instances are blocking, thus they ensure the partial order stated by equations (4-7).

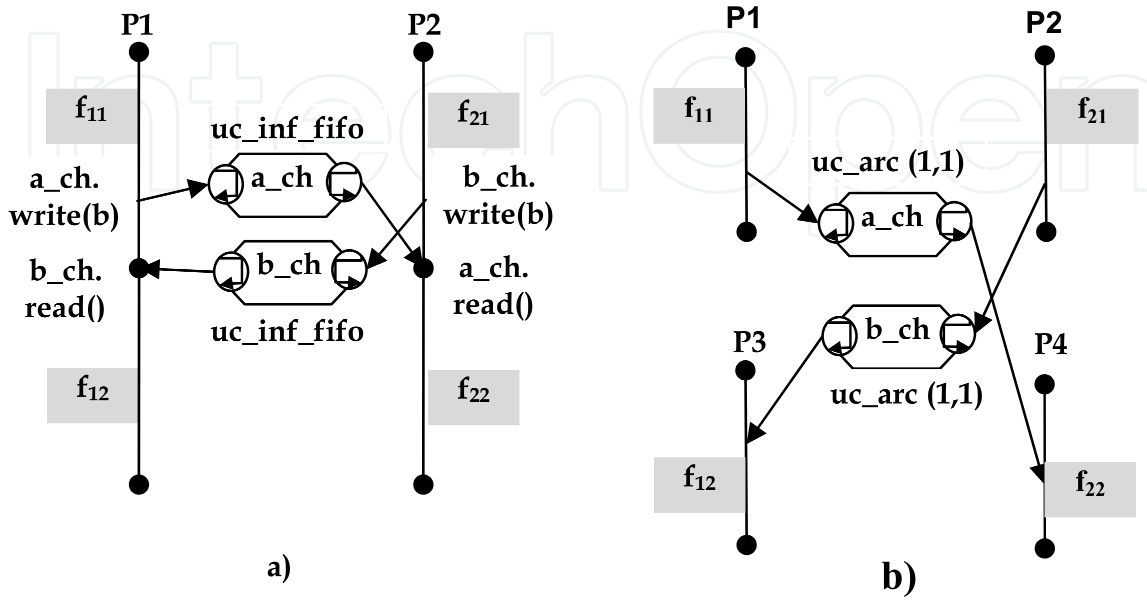


Fig. 10. Specification of Fig.1 solved as a) a Kahn process network and b) as a static dataflow.

Fig.10b shows a second possibility, where the specification is built fulfilling the SDF MoC rules, by using the HetSC methodology and facilities. To fulfil the SDF MoC, the specification style has to be more restrictive than in KPN in several ways. First of all, the KPN specification rules as in the Fig. 10a case, still apply. For instance, only one reader and one writer process can access each channel instance. Furthermore, there are additional rules. For example, each of the specification processes has to be coded without any blocking statement in the middle. Due to this, a single process has been used for each f_{ij} function, enabling a correspondence between a process firing and the execution of function f_{ij}. Moreover, the specific amount of data consumed and produced for each f_{ij} firing has to be known in advance. In HetSC, that information is associated to uc_arc channel instances. The advantage provided by the Fig. 10b solution is that not only does it ensure functional determinism by construction, but it also enables a static analysis based on the extraction of the SDF graph. The Fig. 10b direct SDFG easily leads to the conclusion that the specification is protected against deadlock, and moreover, that a static scheduling is also possible.

5. Conclusions

There is a trade off (shown in qualitative terms in Fig. 11) between the flexibility in the usage of a language and the verification cost for ensuring certain degree of correctness in a specification. In practice, simulation-based methodologies are in the best position for the verification of complex specifications, since formal and semiformal verification techniques easily explode. However, concurrency has become a necessary feature in specification methodologies. Therefore, the capability of simulation based techniques for verification of complex embedded systems has to be reconsidered. A reasonable alternative seems to be the development of cooperative techniques which combine simulation-based methods and specification methodologies which constrain the usage of the language under some formal rules, oriented to fulfilling the desired properties. Specifically, while SystemC is a language with a rich expressivity, it is still necessary to build abstract specification methodologies using SystemC as host language, by constraining the specification facilities and the way they can be used. This way, certain key properties can be guaranteed by construction, and the fulfilment of others can be analyzed. The set of properties to be guaranteed depend on the application domain. Moreover, a formally supported specification methodology can help to validate additional properties through simulation-based verification techniques with a drastic improvement in the detection capabilities and time spent on simulation.

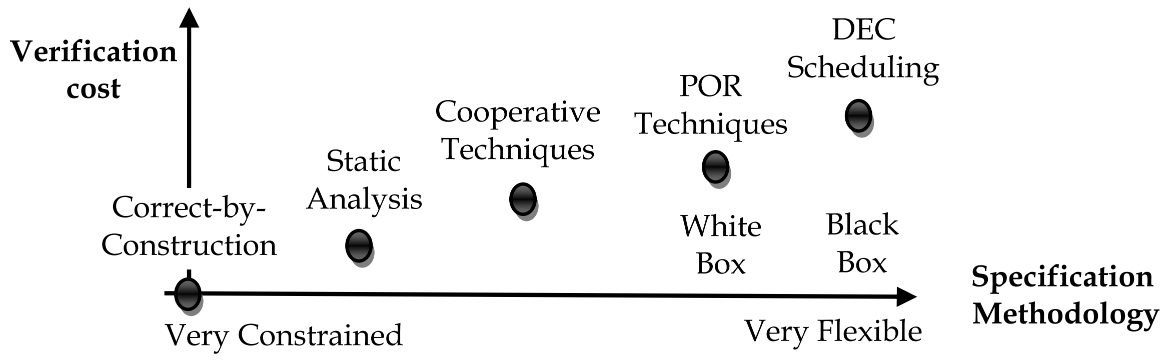


Fig. 11. Trade off between flexibility and verification time after considering concurrency.

6. Acknowledgement

This work has been partially funded by the EU FP7-247999 COMPLEX project and by the Spanish government through the MICINN TEC2008-04107 project.

7. References

Burton, M. et al. (2007). *ESL Design and Verification,* Morgan Kaufman, ISBN 0-12-373551-3

Bergeron, J. (2003) *Writing Testbenches. Functional Verification of HDL Models.* Springer, ISBN 1-40-207401-8.

Cadar, C., Ganesh, V., Pawlowski, P.M., Dill, D.L., & Engler, D.R., (2008). EXE: Automatically Generating Inputs of Death. *ACM Transactions on Information and System Security (TISSEC).* V12, Issue 2, Article 10. December, 2008.

Chiang, S. Y. (2011). Keynote Speech. *Proceedings of ARM Techcom Conference.* October, 25th, 2011. Santa Clara, USA.

EDG website, (2012). EDG Website. http://www.edg.com/. Checked in November, 2011.

Fallah, F., Devadas, S. & Keutzer, K. (1998) Functional vector generation for HDL models using linear programming and 3-satisfiability. *Proceedings of the 35th annual Design Automation Conference (DAC '98)*. ACM, New York, NY, USA, pp. 528-533.

Flanagan, C. & Godefroid, P. (2005) Dynamic Partial Order Reduction for Model Checking Software. *Proceedings of ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 2005.

Godefroid, P. (1995) Partial-Order Methods for the Verification of Concurrent Systems; An approach to the State-Explosion Problem. *PhD thesis*. University of Liege. 1995.

Godefroid, P., Klarlund, N. & Sen, K. (2005) DART: Directed Automated Random Testing. *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation (PLDI '05)*. ACM, New York, NY, USA, pp. 213-223.

Grant, M. (2006). Overview of the MPSoC Design Challenge. *Proceedings of Design Automation Conference 2006, DAC'06.* , ISBN 1-59593-381-6 San Francisco, USA.

Gupta, A., Casavant, A.E., Ashar, P., Mukaiyama, A., Wakabayashi, K. & Liu, X. G. (2002). Property-Specific Testbench Generation for Guided Simulation. *Proceedings of the 2002 Asia and South Pacific Design Automation Conference (ASP-DAC '02)*. IEEE Computer Society, Washington, DC, USA. 2002.

Halfhill, T. (2012). Looking beyond Graphics. 2012. Whipe paper, Available in http://www.nvidia.com/object/fermi_architecture.html.

Haubelt, C., Falk , J., Keinert, J. , Schlichter, T., Streubühr, M. , Deyhle, A. , Hadert, A., Teich, J. (2007). A SystemC-Based Design Methodology for Digital Signal Processing Systems. EURASIP Journal on Embedded Systems. V. 2007, Article ID 47580, 22 pages. January, 2007.

Helmstetter, C. & Maraninchi, F., Maillet-Contoz & Moy, M. (2006) Automatic Generation of Schedulings for Improving the Test Coverage of Systems-on-a-Chip. *Proceedings of Formal Methods in Computer Aided Design, FMCAD'06*. November, 2006.

Helmstetter, C. (2007). Validation de Modèles de Systèmes sur Puce en présence d'ordonnancements Indétermnistes et de Temps Imprecis. *PhD thesis*. March. 2007.

Herrera, F., & Villar, E. (2006). Extension of the SystemC kernel for Simulation Coverage Improvement of System-Level Concurrent Specifications. *Proceedings of the Forum on Specification and Design Languages, FDL'06*. Darmstad. Germany. Sept., 2006.

Herrera, F. & Villar, E. (2007). A Framework for Heterogeneous Specification and Design of Electronic Embedded Systems in SystemC. ACM Transactions on Design Automation of Electronic Systems, Special Issue on Demonstrable Software Systems and Hardware Platforms, V.12, Issue 3, N.22. August, 2007.

Herrera, F., & Villar, E. (2009). Local Application of Simulation Directed for Exhaustive Coverage of Schedulings of SystemC Specifications. *Proc. of the Forum on Specification and Design Languages, FDL'09*. Sophia Antipolis. France. September, 2009. ISBN 1636-9874.

HetSC website, (2012). HetSC website. www.teisa.unican.es/HetSC. 2012.

IEEE, (2005). SystemC Language Reference Manual. Available in http://standards.ieee.org/getieee/1666/download/1666-2005.pdf.

Incisive, (2009). Incisive Enterprise Simulator Datasheet. Available in http://www.cadence.com/rl/Resources/datasheets/incisive_enterprise_specman.pdf. March, 2009

Jantsch, A. (2004). *Modelling Embedded Systems and SoCs. Concurrency and Time in Models of Computation*. Elsevier Science (USA), 2004. ISBN 1-55860-925-3.

Kahn, G. 1974. The Semantics of a simple Language for Parallel Programming. *Proceedings of the IFIP Conference 1974*, North-Holland, 1974.

Kish, L. B. (2002). End of Moore's Law: thermal (noise) death of integration in micro and nano electronics. Physics Letters A 305. pp. 144-149. Elselvier.

Kuhn,T., Oppold, T., Winterholer, M., Rosenstiel, W., Edwards, M. and Kashai, Y. (2001). A Framework for Object Oriented Hardware Specification Verification, and Synthesis. *Proceedings of the Design Automation Conference*, 2001.DAC'01. 2001.

Kundu, S., Ganai, M., Gupta, R. (2008) Partial Order Reduction for Scalable Testing of SystemC TLM Designs. Proceedings of the Design Automation Conference, DAC'08. Anaheim, CA, USA. June, 2008.

Kuo,Y.M., Lin, C.H., Wang, C.Y., Chang, S.H. & Ho, P.H. (2007). Intelligent Random Vector Generator Based on Probability Analysis of Circuit Structure. *Proceedings of the 8th International Symposium on Quality Electronic Design (ISQED '07)*. IEEE Computer Society, Washington, DC, USA, pp. 344-349.

Lee, E. A. & Messerschmitt, D.G. (1987). Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing. *IEEE Transactions on Computers. V. C-36. N.1*. pp. 24-35, January, 1987.

Lee, E.A. (2006). What's the Problem with Threads. *IEEE Computer*, Vol. 36, No. 5, pp. 33-42, May, 2006.

Moy, M., Maraninchi, F., Maillet-Contoz, L. (2005) PINAPA: An Extraction Tool for SystemC Descriptions of Systems on a Chip. *Proceedings of EMSOFT*, September, 2005.

MPI: A Message-Passing Interface Standard. Version 2.2. September, 2009. Available from http://www.mcs.anl.gov/research/projects/mpi/

OSCI Verification WG (2003). SystemC Verification Standard. Version 1.0e. May 16, 2003. Available at www.systemc.org.

OpenMP. (2008). Application Program Interface. 4 Version 3.0 May 2008. Available from http://openmp.org/wp/.

Patel, H.D. & Shukla, S.K. (2004). SystemC kernel extensions for Heterogeneous System Modelling: A Framework for Multi-MoC Modelling and Simulation. Kluwer. 2004.

Sen, K., Marinov, D. & Agha, G.. (2005). CUTE: a Concolic Unit Testing Engine for C. *Proceedings of the 10th European Software Engineering Conference (ESEC/FSE-13)*. ACM, New York, NY, USA, 263-272.

Sen, A., Ogale, V., Abadir, M. S. (2008). Predictive Runtime Verification of multi-processor SoCs in SystemC. Proceedings of Design Automation Conference, DAC'08. Anaheim, CA, USA June, 2008.

UCSCKext, (2011). Website for SystemC kernel extensions provided by University of Cantabria. http://www.teisa.unican.es/HetSC/kernel_ext.html. November, 2011.

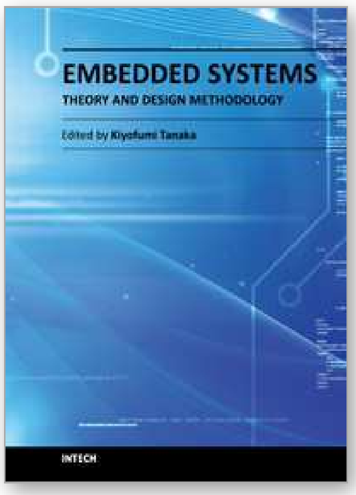
Ugarte, I. & Sanchez, P. (2011) Automatic vector generation guided by a functional metric. *Proceedings of SPIE*. 8067, 80670U (2011)

Yuan, J., Aziz, A., Pixley, C., Albin, K. (2004). Simplifying Boolean constraint solving for random simulation-vector generation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*. V. 23, N. 3, pp. 412-20, March, 2004.

Zhu, J., Sander, I., & Jantsch, A. (2010). HetMoC: heterogeneous modelling in SystemC. Proceedings of Forum for Design Languages (FDL '10). Southampton, UK, 2010.

IntechOpen

IntechOpen



Embedded Systems - Theory and Design Methodology
Edited by Dr. Kiyofumi Tanaka

ISBN 978-953-51-0167-3
Hard cover, 430 pages
Publisher InTech
Published online 02, March, 2012
Published in print edition March, 2012

Nowadays, embedded systems - the computer systems that are embedded in various kinds of devices and play an important role of specific control functions, have permitted various aspects of industry. Therefore, we can hardly discuss our life and society from now onwards without referring to embedded systems. For wide-ranging embedded systems to continue their growth, a number of high-quality fundamental and applied researches are indispensable. This book contains 19 excellent chapters and addresses a wide spectrum of research topics on embedded systems, including basic researches, theoretical studies, and practical work. Embedded systems can be made only after fusing miscellaneous technologies together. Various technologies condensed in this book will be helpful to researchers and engineers around the world.

How to reference
In order to correctly reference this scholarly work, feel free to copy and paste the following:

F. Herrera and I. Ugarte (2012). Concurrent Specification of Embedded Systems: An Insight into the Flexibility vs Correctness Trade-Off, Embedded Systems - Theory and Design Methodology, Dr. Kiyofumi Tanaka (Ed.), ISBN: 978-953-51-0167-3, InTech, Available from: http://www.intechopen.com/books/embedded-systems-theory-and-design-methodology/concurrent-specification-of-embedded-systems-an-insight-into-the-flexibility-vs-correctness-trade-of

INTECH
open science | open minds

InTech Europe
University Campus STeP Ri
Slavka Krautzeka 83/A
51000 Rijeka, Croatia
Phone: +385 (51) 770 447
Fax: +385 (51) 686 166
www.intechopen.com

InTech China
Unit 405, Office Block, Hotel Equatorial Shanghai
No.65, Yan An Road (West), Shanghai, 200040, China
中国上海市延安西路65号上海国际贵都大饭店办公楼405单元
Phone: +86-21-62489820
Fax: +86-21-62489821

© 2012 The Author(s). Licensee IntechOpen. This is an open access article distributed under the terms of the Creative Commons Attribution 3.0 License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.